TCP Byte Counting Refinements

Mark Allman
NASA Glenn Research Center/BBN Technologies
21000 Brookpark Rd. MS 54-2
Cleveland, OH 44135
mallman@grc.nasa.gov

Abstract

TCP's delayed acknowledgment algorithm has been shown to hurt TCP performance. One method of gaining the performance lost by reducing the number of acknowledgments sent is to use a *limited byte counting* algorithm. However, we show that as outlined in [All98], limited byte counting is too aggressive in some situations. This paper defines an *appropriate byte counting* algorithm to fix this aggressiveness. This paper shows that appropriate byte counting is a better overall algorithm. In addition, a scaled version of the appropriate byte counting algorithm, which provides finer-grained control over the aggressiveness of the algorithm, is outlined. In addition, unlike previous work this paper considers the impact of byte counting flows on competing traffic and shows that it is not fundamentally unfair to competing flows that do not use the new algorithm.

1 Introduction

The slow start algorithm [Jac88, APS99] TCP uses to initially determine the available bandwidth of a network path has been the subject of a number of recent research studies. During slow start TCP underutilizes the capacity of the network path, especially if the round-trip time (RTT) between the sender and the receiver is large (such as in satellite networks) [All97]. Several strategies have been proposed to mitigate the impact slow start has on performance.

[Hoe96] observes that the initial slow start period is often terminated by a large number of lost segments, hurting TCP's performance. [Hoe96] suggests setting TCP's *slow start threshold* (*ssthresh*) state variable to the appropriate value, such that slow start ends without causing a large number of dropped segments. Estimating the appropriate value for *ssthresh* is studied further in [AP99]. [AD98] refines the algorithm for estimating *ssthresh* suggested in [Hoe96] and proposes using the estimate to increase the congestion window (*cwnd*), the amount of outstanding data a TCP sender can inject into the network, to half the estimate and pacing segments into the network at an appropriate rate, rather than using the standard slow start algorithm to increase the congestion window.

Another proposal for decreasing the impact of slow start on performance is to use a larger initial value for the congestion window. [APS99] allows the initial congestion window to be 1 or 2 segments[1]. [AFP98] outlines an experimental mechanism for increasing TCP's initial window to 3 or 4 segments (depending on the segment size). Increasing the initial window provides the most benefit for short flows and low bandwidth network paths. Several researchers have studied the impact of using a larger initial window [AHO98, PN98, SP98].

[All98] suggests using a *limited byte counting* (LBC) algorithm to mitigate the impact of delayed acknowledgments on TCP performance. TCP's delayed acknowledgment strategy [Bra89] has been shown to reduce the performance of TCP transfers by slowing the growth of *cwnd* [Pax97, All98, PAD$^+$99]. The standard TCP congestion control algorithms [Jac88, APS99] call for a TCP sender to increase the size of *cwnd* by a single segment for each acknowledgment (ACK) received during slow start. A receiver generating delayed ACKs reduces the number of returning ACKs by approximately half when compared to a receiver that generates an ACK for each segment received. Therefore, the rate that *cwnd* is increased is also reduced. Using LBC, the amount *cwnd* is increased is based on the amount of new data covered by each incoming ACK, rather than being a constant.

As defined in [All98], LBC's *cwnd* increase during slow start based loss recovery is slightly too aggressive. We slightly alter the LBC algorithm and define an *appropriate byte counting* algorithm that does not behave too aggressively during loss recovery. In some networks byte counting, as defined in [All98] and in section 3, may be too aggressive, yet the standard algorithm may not be aggressive enough. Therefore, we briefly introduce a scaled version of byte counting that provides a way to control the aggressiveness of the algorithm. [All98] lacks a discussion of the impact of byte counting flows on transfers not utilizing the byte counting algorithm. However, judging the fairness of the algorithm is important from the perspective of deploying byte counting in real networks. Therefore, we present three preliminary experiments that help gauge the impact a byte counting transfer has on non-byte counting flows.

This paper is organized as follows. Section 2 outlines the network layout for the simulations presented in this paper. Section 3 outlines the *appropriate byte counting* algorithm and presents simulations showing the behavior of the algorithm. Section 4 investigates adding a scale factor to allow

[1]In practice, most TCP implementations maintain the congestion window in terms of bytes, rather than segments. However, to simplify the discussion we discuss the congestion window in terms of segments in this paper. The conversion between segments and bytes is straightforward.

finer-grained control of the byte counting algorithm. Section 5 studies the fairness implications of using appropriate byte counting. Finally, section 6 outlines our conclusions and provides a discussion of future work in this area.

2 Simulation Environment

We used the *ns* [MF95] network simulator to conduct the experiments presented in this paper. The entire network layout for our simulations is given in Figure 1. The sender, *S*, transmits data to the receiver, *R*, via a gateway, *G*. The bottleneck bandwidth is 1.5 Mbps (approximately T1 rate). The length of the queue in the gateway is 40 segments (approximately twice the *delay-bandwidth* product of the network path). Some of the simulations presented in this paper use RED queueing [FJ93, BCC$^+$98]. The RED parameters used in those experiments are given in table 1.

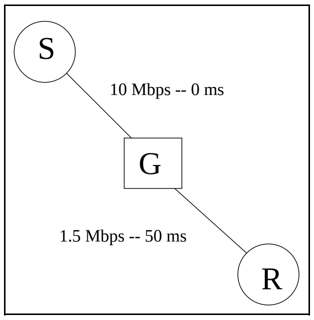

Figure 1: Simulated network topology used in all simulations presented in this paper.

Parameter	Value
min_{th}	7
max_{th}	21
w_q	0.002
max_p	0.1

Table 1: RED parameters.

The standard TCP Reno [Jac88, APS99] and TCP SACK [MMFR96, FF96] modules provided in *ns* are used for the baseline tests, and modifications of these modules are used to investigate byte counting. The segment size used in these tests was 1000 bytes. In addition, all TCP flows used in these simulations employed an initial congestion window of 2 segments (currently allowed by [APS99] and discussed in [AFP98]). The advertised window size was 20 segments (roughly the *delay-bandwidth* product of the network).

3 Appropriate Byte Counting

Currently, the slow start algorithm increases TCP's *congestion window* (*cwnd*) by a constant amount for each acknowledgment (ACK) received [APS99]. [All98] investigates the performance of a limited byte counting (LBC) algorithm for increasing *cwnd* based on the number of bytes acknowledged, rather than by one segment for each ACK received. The idea behind the LBC algorithm is that the sender should not reduce the rate of *cwnd* growth based on whether or not the receiver generates delayed acknowledgments. However, blindly incrementing *cwnd* based on the number of new data bytes acknowledged upon receipt of each ACK is overly aggressive during slow start based loss recovery, as well as in the face of stretch ACKs (acknowledgments covering more than 2 segments of previously unacknowledged data). The remedy suggested in [All98] is to place an upper bound of 2 segments on the increase of *cwnd* in response to a single ACK. This allows *cwnd* to open in a similar manner regardless of whether or not the receiver implements delayed acknowledgments, but not open *cwnd* in a very over-aggressive manner during loss recovery.

However, even using the limit provided by LBC, byte counting is slightly more aggressive than the standard algorithm during slow start based loss recovery. An ACK for *N* previously unacknowledged segments during loss recovery does not indicate that *N* segments have left the network, as such an ACK indicates during a connection's initial slow start period. The only accurate determination that can be made when an ACK arrives during slow start based loss recovery is that at least one segment has left the network[2]. In addition, [APS99] specifies that a TCP receiver *should not* delay ACKs during loss recovery (i.e., ACKs for out-of-order segments). So, if delayed ACKs are not generated LBC is not needed to ameliorate their impact. However, as defined in [All98], LBC will attempt to counteract the delayed ACKs anyway and end up being too aggressive. Therefore, we define *appropriate byte counting* (ABC) as using limited byte counting only during the initial slow start period[3]. Also note that the behavior of ABC (or LBC) is nearly identical to the standard behavior when the receiver is generating an ACK for each incoming segment. Only in the case of ACK loss is the behavior different (the ABC algorithm will increase *cwnd* more than the standard algorithm). However, given that the difference is slight we did not experiment with ABC in conjunction with receivers that ACK each segment.

Figures 2 and 3 show the behavior of ABC in comparison to that of LBC and the standard algorithm under various amounts of network load for TCP SACK[4] flows. Drop-tail queueing was used in these experiments. In these figures, the "ACK Every Segment" line denotes a standard TCP sender communicating with a receiver that generates an ACK for each arriving segment. The "Delayed ACKs" line shows the behavior of a standard TCP sender transmitting data to a receiver implementing delayed ACKs. The "Appropriate Byte Counting" and "Limited Byte Counting" lines show the behavior of ABC and LBC senders when communicating with a delayed ACK receiver. In these simulations, a given number of random length flows (shown on the *x*-axis) were generated at a random starting time during the simulation period (100 seconds). Each point on the plots represents the average of 30 different random scenarios. While these scenarios are not necessarily representative of real-world traffic patterns, they provide insights into the behavior of the algorithms under a varying amount of network load.

Figure 2 shows the throughput (unique data bytes per sec-

[2]A TCP implementation may be able to use the received SACK blocks to derive better information about the number of segments being acknowledged.

[3]ABC can also be used during the slow start that follows a long idle period in some TCP implementations.

[4]For this set of experiments, the TCP Reno behavior was very similar to the TCP SACK behavior shown and therefore we omitted the TCP Reno plots for brevity.

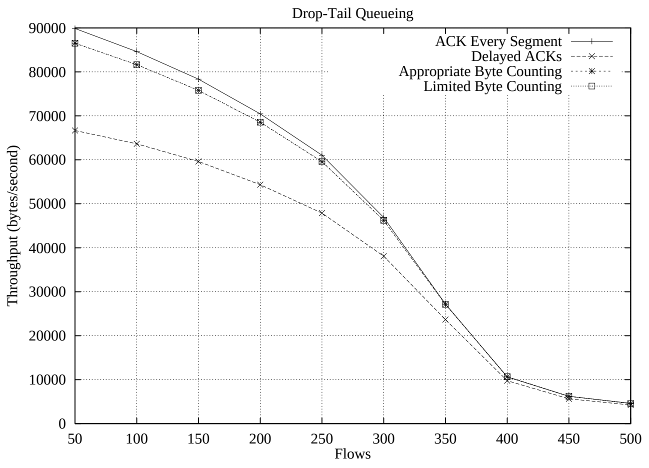

Figure 2: Impact on throughput of ABC on SACK-based flows over a network utilizing a drop-tail queue.

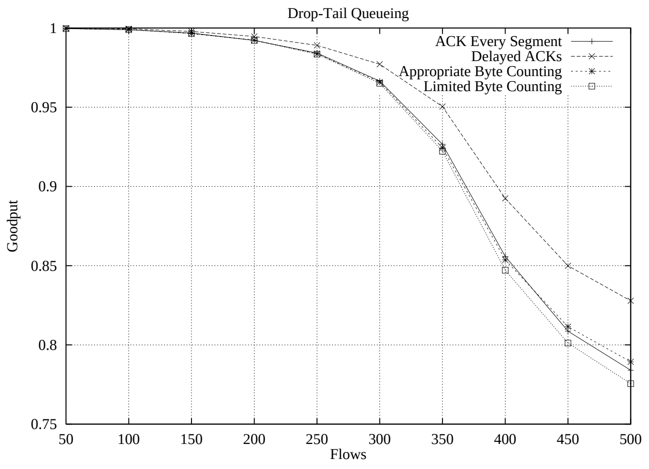

Figure 3: Impact on goodput of ABC on SACK-based flows over a network utilizing a drop-tail queue.

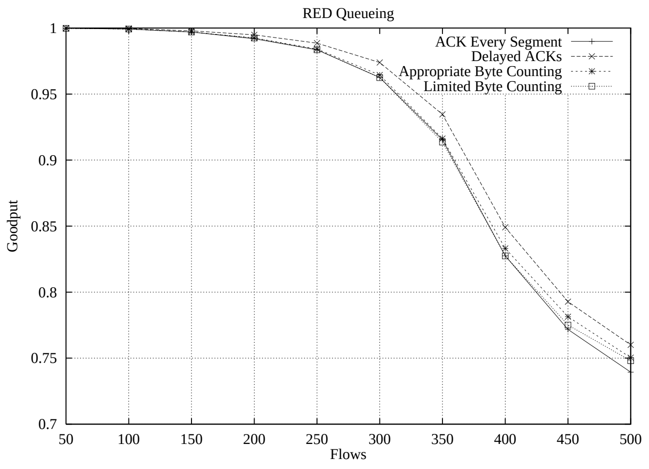

Figure 4: Impact on goodput of ABC on SACK-based flows over a network utilizing a RED queue.

ond) as a function of the number of flows generated during the simulation. First, using the standard algorithm with delayed ACKs provides the lowest throughput, due to the less aggressive nature of this variant when compared to the others studied. The figure shows that the throughput benefit of using ABC is nearly identical to that gained by using LBC. Furthermore, using ABC (or LBC) provides much better throughput than the standard algorithm in the face of delayed acknowledgments. The throughput gained by an ABC/LBC sender when faced with delayed ACKs is almost as high as the throughput gained when using the standard algorithm and receiving an ACK for each segment sent. The ABC/LBC algorithms coupled with delayed ACKs make a TCP sender slightly more bursty than the standard algorithm when receiving an ACK for each segment, which explains the discrepancy in throughput [All98]. This figure indicates that using ABC can achieve nearly the same throughput benefit as eliminating delayed ACKs, while not increasing the amount of network traffic (by roughly half as many ACKs). In addition, this figure shows that there is nearly no performance hit for using ABC over LBC.

Figure 3 shows the goodput[5] as a function of the number of flows generated during each simulation. The plot shows that using the standard *cwnd* increase algorithm with delayed ACKs achieves the best goodput. In this situation, a TCP sender is not nearly as aggressive as the other three versions of TCP shown in the plot. Therefore, it is expected that the less aggressive TCP variant also experiences the least amount of loss. The figure shows that LBC provides lower goodput than the standard algorithm when receiving an ACK for each segment. However, ABC shows slightly better goodput than the standard algorithm when each segment is ACKed. The discrepancy between LBC and ABC shows that the extra segments sent during recovery by an LBC sender are largely unnecessary for loss recovery. That is, each ACK during slow start based loss recovery generally points to a segment that has been lost in the network. Transmitting segments other than the one pointed to by the incoming ACK may be unnecessary. Therefore, increasing the number of these possibly unnecessary segments generally decreases the goodput of the transfer. ABC reverts to the standard *cwnd* increase algorithm during loss recovery via slow start and therefore does not send the extra segments transmitted by LBC and therefore shows better goodput.

Simulations with RED queueing in the congested gateway show throughput averages similar to those shown in figure 2. The goodput for the tests conducted with RED queueing is shown in Figure 4. The goodput in the RED environment is less than in the drop-tail for all versions of TCP tested due to RED's early dropping mechanism. In other words, RED increases the drop rate for the network but in doing so reduces the average queue size so that the gateway is not biased against bursty senders. The figure illustrates both these facets of RED queueing. The figure shows that ABC still achieves slightly better goodput than LBC indicating that LBC is sending unnecessary segments during slow start style loss recovery.

4 Scaled Byte Counting

In some cases ABC may be too aggressive while the standard algorithm coupled with delayed ACKs may not be aggressive enough. In this section we explore a *scaled* version of ABC (SABC) that can be used to further tune the algorithm. During slow start, SABC increments *cwnd* according to Equation 1, where N is the number of previously unacknowledged segments covered by the incoming ACK and S is the *scale factor.*

$$cwnd = cwnd + 1 + ((N - 1) \cdot S) \qquad (1)$$

The congestion window is first incremented by a single segment for the data segment that triggered the transmission of the ACK, per the standard algorithm. Then, *cwnd* is further incremented by some fraction of a segment for each additional newly acknowledged segment covered by the ACK. Setting $S = 0$ makes this algorithm identical to the standard algorithm (i.e., no credit for more than one segment per ACK). Setting $S = 1$ yields unlimited byte counting (i.e., we increment *cwnd* by 1 full segment for each new segment acknowledged). Making $S > 1$ would violate the ideas presented in [Jac88] because *cwnd* would more than double every RTT during slow start. SABC is only used during the first slow start period (i.e., not during loss recovery via slow start). For example, consider the case when $S = 0.5$ and the TCP sender receives two ACKs, each for two outstanding segments. Also, assume the current *cwnd* value is C. Upon receiving the first ACK the sender will increment *cwnd* to $C = C + 1 + 0.5 = C + 1.5$ and 3 segments will be transmitted (2 due to the window sliding and 1 due to the *cwnd* increase). Additionally, the sender will have 0.5 segments left in the congestion window, which will remain unused (at this point) since TCP sends only when it can transmit a full segment. The next ACK will again increase the congestion window by 1.5 segments. However, this time TCP will transmit 4 segments (2 for the window sliding and 2 for the *cwnd* increase).

Figures 5 and 6 show the costs and benefits of SABC with several choices for S in a drop-tail queueing environment. The TCP used for these simulations utilizes the TCP SACK option. The traffic patterns used in these simulations are the same scenarios used in the previous section, with each point again representing the average of 30 random scenarios. As expected, Figure 5 shows that as we increase S (and therefore, make the TCP sender more aggressive) we decrease the average goodput. However, Figure 6 shows that as we increase S, TCP is able to attain better throughput on average. Simulations of Reno-based TCP yield similar results. Tests involving RED queues show approximately the same results with respect to throughput. However, the difference in the drop rate among the various values of S is diminished with RED queueing due to RED's ability to handle bursts better than drop-tail queues, as shown in Figure 7. These figures confirm that using a scaled version of the ABC algorithm allows finer-grained control over the aggressiveness of the algorithm. Further tests in real networks are needed to determine if this additional control is useful.

5 Fairness of Byte Counting

In order to deploy a new TCP mechanism in the global Internet, it must be fair to traffic that does not employ the new algorithm. This section presents several experiments aimed at assessing whether TCP with the ABC algorithm competes fairly against non-ABC traffic, or whether the algorithm steals resources from non-ABC transfers.

5.1 One-on-One Tests

To investigate the impact one ABC transfer has on one non-ABC transfer we constructed a set of simulations consisting of one long TCP flow and one relatively short TCP

[5]Goodput is defined as the ratio of the number of unique data bytes sent to the total number of data bytes sent.

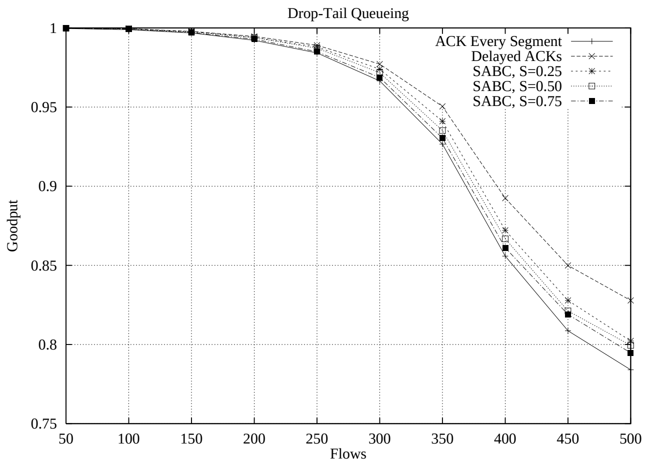

Figure 5: Impact on goodput of SABC on SACK-based flows over a network utilizing a drop-tail queue.

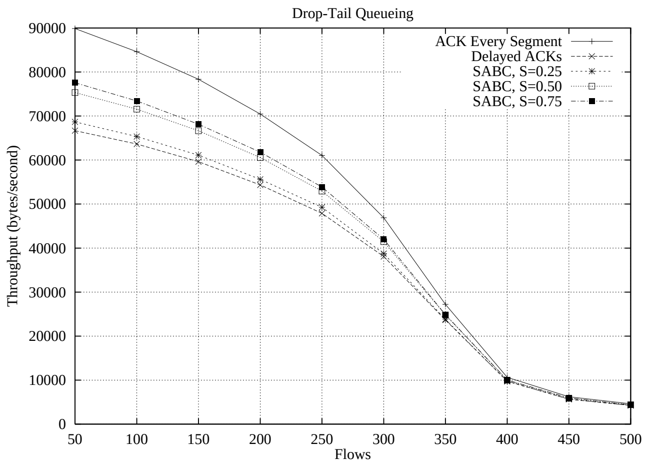

Figure 6: Impact on throughput of SABC on SACK-based flows over a network utilizing a drop-tail queue.

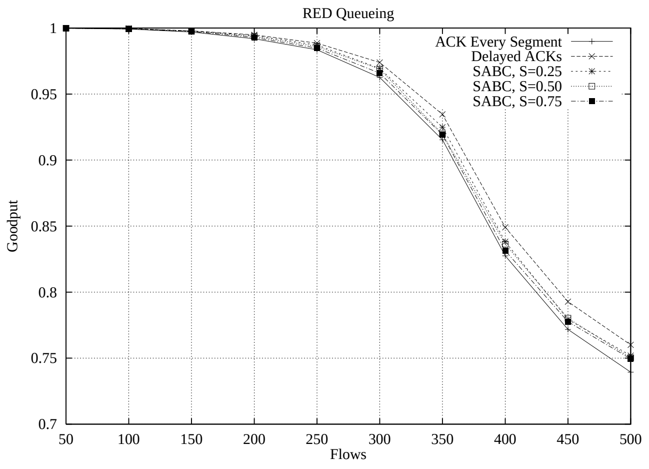

Figure 7: Impact on goodput of SABC on SACK-based flows over a network utilizing a RED queue.

Long/Short	Throughput Ratio	Dropped Segment Ratio
DA/DA	1.00/1.00	1.00/1.00
DA/ABC	1.00/1.09	1.02/1.94
ABC/DA	1.00/0.91	1.97/1.08
ABC/ABC	0.99/1.09	1.98/2.42

Table 2: One-on-One Tests: TCP SACK, Drop-Tail Queueing. Each reported value is the ratio between the experimental transfer and a similar transfer when delayed ACKs were used for all connections.

flow. The short flow (5,000,000 bytes) was initiated and completed within the time required to complete the long transfer (20,000,000 bytes). The simulation was repeated 30 times with random start times for the short transfer (however, the short transfer always ended before the long transfer).

Table 2 shows the one-on-one simulation results with a drop-tail queue in the congested gateway and SACK-based TCP. The first column of the table reports the TCP variants used for the long and short transfers (with "DA" denoting a sender using the standard *cwnd* increase algorithm and a receiver that generates delayed ACKs). The second column gives the throughput ratio for both transfers between the scenario in question and the simulations utilizing delayed ACKs and the standard *cwnd* increase algorithm for both transfers. The third column reports the ratio of number of dropped segments when compared to the standard *cwnd* increase algorithm in the face of delayed ACKs for both transfers.

The table shows that throughput for the long transfer is stable across all combinations of TCP variants presented. In addition, introducing ABC into one or both flows does not change the throughput of the short flow by more than 10% (up or down). In addition, the simulations show that the more aggressive ABC flows do not drastically increase the drop rate of the competing DA flows, even though the overall drop rate for the network is increasing.

Table 3 shows the behavior of SACK-based TCP variants in a RED queueing environment. This table shows that the throughput obtained by the long transfer is stable regardless of whether ABC is utilized by the sender. Furthermore, when the short transfer is made using ABC the throughput increases, when compared to a short DA flow. However, when ABC is used for the long flow and DA for the short transfer, the DA flow obtains less throughput than it does when competing against a long DA flow. The long flow (ABC, in this case) is not stealing the bandwidth from the short flow, as the reported throughput for the long ABC flow is similar in each simulation reported. However, the ABC flow increases the delay along the network path (by increasing the queue length) and therefore it takes the DA flow longer to obtain an appropriate congestion window, leading to lower throughput.

5.2 Several Concurrent Connections

The next experiment performed to judge the fairness of ABC when competing with non-ABC transfers is to run a set of 16 parallel TCP connections across the network. The 3,000,000 byte transfers are started at the same time. We expect that if the TCP algorithms are completely "fair", each will receive the same amount of the bottleneck bandwidth. We used the *fairness index* [Jai91] given in Equation 2 to quantify the fairness of the set of connections.

$$f(x_1, x_2, \cdots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \cdot \sum_{i=1}^{n} x_i^2} \qquad (2)$$

When given a set of non-negative throughputs (x_1,x_2, etc.) Equation 2 yields a value between 0 and 1. If all n throughputs are the same the equation yields 1. If m of the n throughputs are the same, with the remaining $n - m$ connections receiving zero throughput the equation yields m/n. Note that these simulations are not necessarily realistic, but do provide insight into an algorithm's fundamental ability to share a congested link in a very simple situation.

Table 4 shows results from various simulations constructed to test ABC's fairness. All simulations consisted of 16 TCP connections, as described above. However, the number of flows utilizing each variant of TCP explored in this paper is varied. The table indicates how many flows used each TCP variant tested, as well as the fairness index for each simulation. The scenarios presented were constructed to show the fairness behavior in a range of situations (i.e., some Reno TCP connections, some SACK TCP connections, etc.). The table shows that in both drop-tail and RED queueing environments and with both Reno and SACK TCP, ABC shares the bandwidth as well as either the standard increase algorithm when receiving an ACK for every segment (AE) or when receiving delayed acknowledgments (DA) (i.e., the fairness index yields a value very close to 1).

5.3 Many Concurrent Connections

Finally, we tested the fairness of ABC in the context of a random traffic mix. As in sections 3 and 4, we used 30 random scenarios consisting of a variable number of TCP flows (50–500) that were started at random times throughout the course of the simulation period (100 seconds). We conducted these tests with SACK-based TCP flows. Half the flows in each run utilized the standard *cwnd* increase algorithm, with the other half of the connections using ABC. The receiver generated delayed ACKs for all flows. Figures 8 and 9 show the results of these experiments when utilizing a drop-tail queue in the congested gateway. The two lines marked "Pure Delayed ACKs" and "Pure ABC" are the average throughput for runs in which all connections utilized the standard algorithm or the ABC algorithm (as reported in section 3). The "Mixed Delayed ACK" line indicates the average goodput or throughput of the connections utilizing the standard *cwnd* increase algorithm in the mixed traffic simulation. Likewise, the "Mixed ABC" line shows the average goodput or throughput of the ABC flows in the mixed traffic environment.

Long/Short	**Throughput Ratio**	**Dropped Segment Ratio**
DA/DA	1.00/1.00	1.00/1.00
DA/ABC	1.00/1.15	1.07/1.71
ABC/DA	1.01/0.90	1.69/1.03
ABC/ABC	1.01/1.01	1.74/1.67

Table 3: One-on-One Tests: TCP SACK, RED Queueing. Each reported value is the ratio between the experimental transfer and a similar transfer when delayed ACKs were used for all connections.

Queue Type	**Reno AE**	**Reno DA**	**Reno ABC**	**SACK AE**	**SACK DA**	**SACK ABC**	**Fairness Index**
RED	0	0	0	16	0	0	0.9980
RED	0	0	0	0	16	0	0.9992
RED	0	0	0	0	0	16	0.9990
RED	0	0	0	0	8	8	0.9945
RED	0	0	0	8	0	8	0.9979
RED	0	0	0	4	6	6	0.9955
RED	16	0	0	0	0	0	0.9984
RED	0	16	0	0	0	0	0.9991
RED	0	0	16	0	0	0	0.9981
RED	0	8	8	0	0	0	0.9912
RED	8	0	8	0	0	0	0.9980
RED	4	6	6	0	0	0	0.9929
RED	0	4	4	0	4	4	0.9928
Drop-Tail	0	0	0	16	0	0	0.9993
Drop-Tail	0	0	0	0	16	0	0.9968
Drop-Tail	0	0	0	0	0	16	0.9986
Drop-Tail	0	0	0	0	8	8	0.9973
Drop-Tail	0	0	0	8	0	8	0.9974
Drop-Tail	0	0	0	4	6	6	0.9976
Drop-Tail	16	0	0	0	0	0	0.9985
Drop-Tail	0	16	0	0	0	0	0.9932
Drop-Tail	0	0	16	0	0	0	0.9940
Drop-Tail	0	8	8	0	0	0	0.9985
Drop-Tail	8	0	8	0	0	0	0.9880
Drop-Tail	4	6	6	0	0	0	0.9943
Drop-Tail	0	4	4	0	4	4	0.9865

Table 4: Jain's Fairness Index

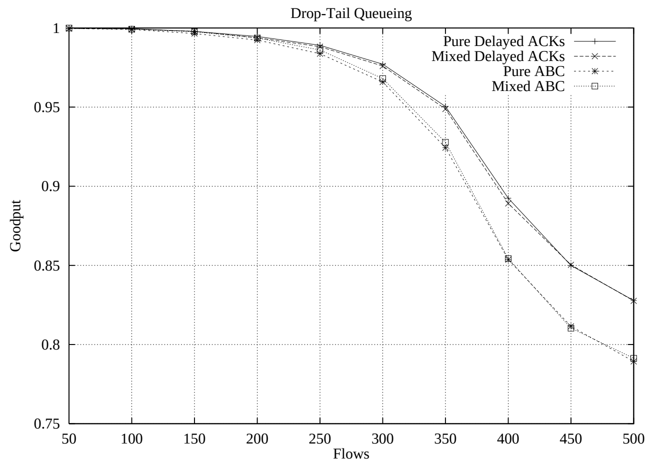

Figure 8: Impact on goodput of ABC flows in a mixed traffic environment. TCP SACK-based flows are used over a network utilizing a drop-tail queue.

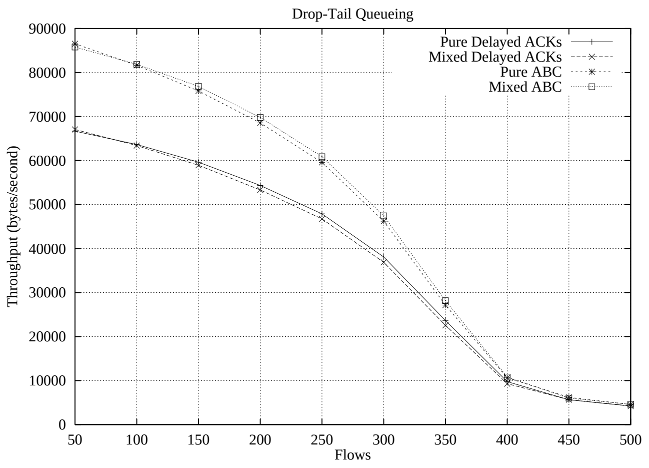

Figure 9: Impact on throughput of ABC flows in a mixed traffic environment. TCP SACK-based flows are used over a network utilizing a drop-tail queue.

As Figure 8 shows that both versions of TCP obtain nearly the same goodput regardless of which *cwnd* increase algorithm is being used in the competing traffic. Figure 9 shows that each variant of TCP receives approximately the same throughput regardless of which version of TCP it is competing against. However, there is a very slight throughput advantage for ABC when competing against the standard *cwnd* increase algorithm in a number of the simulations. Utilizing RED queueing in the congested gateway yields similar results.

6 Conclusions and Future Work

The following are the conclusions we can draw from the simulations presented in this paper, as well as some areas for future work.

- ABC is more appropriate than LBC during loss recovery via slow start, due to the unnecessary extra segments transmitted by LBC. Furthermore, ABC increases throughput, as well as the drop rate in a similar manner to turning off delayed ACKs without introducing more traffic into the network. However, ABC still needs to be tested over a wide range of real network paths. One area of possible concern may be in increasing the burstiness of TCP in network paths with a very small number of buffers available.
- SABC provides a finer-grained control over the increase of *cwnd* and may provide a way to tune the algorithm if ABC turns out to be overly aggressive in real network experiments. SABC shows more promise in networks with drop-tail gateways, and not so much in RED environments (due to RED's ability to handle small bursts better than drop-tail gateways).
- ABC does not show any fundamental bias against competing traffic in the simulations presented in this paper.
- Finally, the simulations presented in this paper offer a glimpse of how ABC is likely to perform. However, experiments conducted over the Internet are required before ABC can be judged to be safe for widespread implementation.

Acknowledgments

This paper has benefited from conversations with a great number of people. I would like to thank the following people for their constructive discussions on byte counting: Sally Floyd, Dan Glover, Chris Hayes, Matt Mathis and Jeff Semke. In addition, Shannon Steinfadt and the anonymous CCR reviewers provided useful comments on an earlier version of this paper. My thanks to all!

References

[AD98] Mohit Aron and Peter Druschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. Technical Report TR98-318, Rice University Computer Science, 1998.

[AFP98] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's Initial Window, September 1998. RFC 2414.

[AHO98] Mark Allman, Chris Hayes, and Shawn Ostermann. An Evaluation of TCP with Larger Initial Windows. *Computer Communication Review*, 28(3), July 1998.

[All97] Mark Allman. Improving TCP Performance Over Satellite Channels. Master's thesis, Ohio University, June 1997.

[All98] Mark Allman. On the Generation and Use of TCP Acknowledgments. *Computer Communication Review*, 28(5), October 1998.

[AP99] Mark Allman and Vern Paxson. On Estimating End-to-End Network Path Properties. In *ACM SIGCOMM*, September 1999. To appear.

[APS99] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control, April 1999. RFC 2581.

[BCC$^+$98] Robert Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and Lixia Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet, April 1998. RFC 2309.

[Bra89] Robert Braden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.

[FF96] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3), July 1996.

[FJ93] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[Hoe96] Janey Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *ACM SIGCOMM*, August 1996.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. Wiley, 1991.

[MF95] Steven McCanne and Sally Floyd. NS (Network Simulator), 1995. URL http://www-nrg.ee.lbl.gov.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.

[PAD$^+$99] Vern Paxson, Mark Allman, Scott Dawson, William Fenner, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke, and Bernie Volz. Known TCP Implementation Problems, March 1999. RFC 2525.

[Pax97] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM*, September 1997.

[PN98] Kedarnath Poduri and Kathleen Nichols. Simulation Studies of Increased Initial TCP Window Size, September 1998. RFC 2415.

[SP98] Tim Shepard and Craig Partridge. When TCP Starts Up With Four Packets Into Only Three Buffers, September 1998. RFC 2416.